The NetLogger Methodology for Distributed System Performance Analysis

Dan Gunter, Brian Tierney, Brian Crowley, Mason Holding, Jason Lee
Lawrence Berkeley National Laboratory

Abstract

Diagnosis and debugging of performance problems on complex distributed systems requires access to performance information at both the application and system level from a heterogeneous collection of hosts and networks. We describe a methodology, called NetLogger, that enables real-time diagnosis of performance problems in such systems. The methodology includes tools for generating precision event logs, an interface to a Java-based agent system that automates the execution of monitoring sensors and collection of monitoring events, and tools for visualizing the log data and real-time state of the distributed system. The approach is novel in that it combines network, host, and application-level monitoring, providing a complete view of the entire system.

1. Introduction

The performance characteristics of distributed applications are complex, rife with "soft failures" in which the application produces correct results but has much lower throughput or higher latency than expected. Because of the complex interactions between multiple components in the system, the cause of the performance problems is often elusive. Bottlenecks can occur in any component along the data's path: applications, operating systems, device drivers, network adapters, and network components such as switches and routers. Sometimes bottlenecks involve interactions between components, sometimes they are due to unrelated network activity impacting the distributed system.

Usually the interactions between components are not known ahead of time, and may be difficult to replicate. Therefore, it is important to capture as much of the system behavior as possible while the application is running. It is also important to respond to performance problems as soon as possible; while post-hoc diagnosis of this data is valuable for systemic problems, for operational problems users will have already suffered through a period of degraded performance.

We have developed a methodology, called NetLogger, for monitoring, under realistic operating conditions, the behavior of all elements of the application-to-application communication path in order to determine exactly what is happening within a complex system.

Distributed application components are modified to produce timestamped logs of "interesting" events at all the critical points of the distributed system. The events are correlated with the system's behavior in order to characterize the performance of all aspects of the system and network in detail

Monitoring of the system's behavior can also be modified dynamically while an application is running. A separate system of software agents, called Java Agents for Monitoring and Management (JAMM), provides a coherent and flexible interface to an extensible set of system "sensors", which provide essential data such as CPU load, interrupt rate, TCP retransmissions, TCP window size, and so on. Visualization of the log data can be used to provide interactive feedback to the system monitoring service.

NetLogger has demonstrated its usefulness in several contexts, including the Distributed Parallel Storage System (DPSS)[ref], and Radiance[ref]. Both of these are loosely-coupled client-server architectures. In principle, however, the approach is adaptable to any distributed system architecture. The way in which NetLogger is integrated into a distributed system will vary, but NetLogger's behavior and utility are independent of any particular system design.

2. NetLogger Toolkit Components

All the tools in the NetLogger Toolkit share a common log format, and assume the existence of accurate and synchronized system clocks. The NetLogger Toolkit itself consists of three components: an API and library of functions to simplify the generation of application-level event logs, a set of tools for collecting and sorting log files, and a tool for visualization and analysis of the log files.

2.1. Common log format

NetLogger uses the IETF draft standard Universal Logger Message format (ULM) [ref] for the logging and exchange of messages. Use of a common format that is plain ASCII text and easy to parse simplifies the processing of potentially huge amounts of log data, and makes it easier for third-party tools to gain access to the data.

The ULM format consists of a whitespace-separated list of “field=value” pairs. ULM required fields are DATE, HOST, PROG, and LVL; these can be followed by any number of user-defined fields. NetLogger adds the field NL.EVNT, whose value is a unique identifier for the event being logged. The value for the DATE field has six digits of accuracy, allowing for microsecond precision in the timestamp. Here is a sample NetLogger ULM event:

DATE=20000330112320.957943 HOST=dpss1.lbl.gov PROG=testProg LVL=Usage NL.EVNT=WriteData SEND.SZ=49332

This says that the program *testprog* on host *dpss1.lbl.gov* performed a *WriteData* event with a send size of 49,322 on March 30, 2000 at 11:23 (and some seconds) in the morning.

The user-defined events at the end of the log entry can be used to record any descriptive value or string that relates to the event such as message sizes, non-fatal exceptions, counter values, and so on.

2.2. Clock synchronization

In order to analyze a network-based system using absolute timestamps, the clocks of all relevant hosts must be synchronized. This can be achieved using a tool which supports the Network Time Protocol (NTP) [ref], such as the *xntpd* [ref] daemon. By installing a GPS-based NTP server on each subnet of the distributed system and running *xntpd* on each host, all the hosts’ clocks can be synchronized to within about 0.25ms. If the closest time source is several IP router hops away, accuracy may decrease somewhat. However, it has been our experience that synchronization within 1 ms is accurate enough for many types of analysis. The NTP web site [ref] has a list of public NTP servers that one can connect to and synchronize with.

2.3. NetLogger API

In order to instrument an application to produce event logs, the application developer inserts calls to the NetLogger API at all the critical points in the code, then links the application with the NetLogger library. This facility is available in six languages: Java, C, C++, Perl, Python, and Fortran. The API has been kept as simple as possible, while still providing automatic timestamping of events and logging to either a local file, syslog, or to a remote host.

Here is a sample of the Java API usage:

```
NetLogger eventLog = new NetLogger(“testprog”);
eventLog.open( “dolly.lbl.gov”, 14830 );
…
eventLog.write(“WriteData”, “SEND.SZ=” + sz );
…
eventLog.close();
```

If the value for *sz* is `49332`, and the program is running on the host `dpss1.lbl.gov,` the *write()* statement above will produce the sample log entry provided in the description of ULM, above. In this case, the data will be sent to port 14830 on the host dolly.lbl.gov.

Even in a compiled language such as C, logging can perturb the host or the network if it occurs too frequently. In order to accommodate short, sparse bursts of activity, the library can optionally buffer the log entries in memory, and later flush them from the system. We have found that network load is not greatly affected if less than one thousand messages per second are logged, however, an internal binary format, such as Pablo’s SDDF [ref] is being actively researched as a way to extend NetLogger to handle higher-throughput scenarios.

2.4. Event log collection and sorting

NetLogger facilitiates the collection of event logs from an application which runs across a wide-area network by providing automatic logging to a chosen host and port. A server daemon, called *netlogd*, receives the log entries and writes them into a file on the local disk. Thus, applications can transparently

log events in real-time to a single destination over the wide-area network. A similar tool, the *Real-Time Collector*, has been developed in order to receive events from both applications and system event monitoring services, such as the JAMM monitoring system; discussion of this tool will be deferred until after JAMM has been presented and explained.

Event log entries are written in the order in which they arrive, which may not necessarily be in timestamp order, or in the order most useful to the user. Therefore, a program called *nlsort* has been written which can re-order ULM event logs by timestamp or a combination of user-defined fields.

2.5. Event log visualization and analysis

We have found exploratory, visual analysis of the log event data to be the most useful means of getting at the causes performance anomalies. The NetLogger Visualization tool, *nlv*, has been developed to provide a flexible and interactive graphical representation of system-level and application-level events. *Nlv* uses three types of graph primitives to represent different events. These are shown in Figure 1.

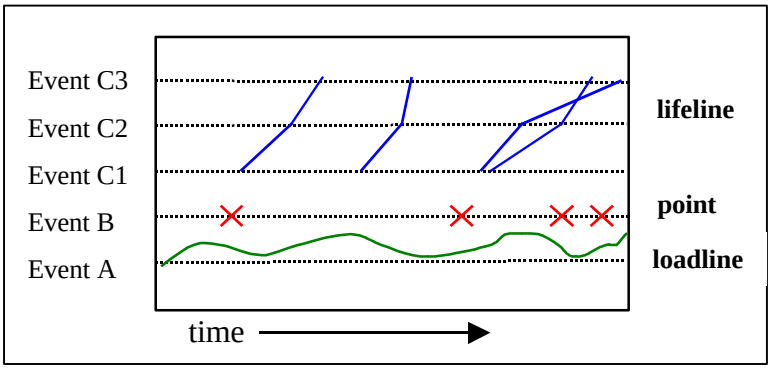

Figure 1: *nlv* Graph Primitives

The most important of these primitives is the *lifeline*, which represents the "life" of an *object* (datum or computation) as it travels through a distributed system. With time shown on the x-axis, and ordered events shown on the y-axis, the slope of the lifeline gives a clear visual indication of the time it took for the object to move through the distributed system. Each object is given a unique identifier by placing a unique combination of values in one or more of its ULM fields. These values are used for all events along the path. In a client-server system, one such event path might include: a request's dispatch from the client, the request's arrival at the server, the begin and end of server processing of the request, the response's dispatch from the server, and the response's arrival at the client.

The other two graph primitives are the *loadline* and the *point*. The loadline connects a series of scaled values into a continuous segmented curve, and is most often used for representing the rise and fall of system resources such as CPU load or free memory. The point data type is used to graph single occurrences of events, often error or warning conditions such as TCP retransmits.

In order to assist correlation of observed system performance with logged events, *nlv* has been designed to allow real-time visualization of the event data as well as historical browsing and playback of interesting time periods. In the real-time mode, the graph scrolls along the time axis (x-axis) in real time, showing data as it arrives in the event log. In historical mode, the user can change the position in the log file, change the scale of the graph, zoom in and out interactively, choose a subset of events to look at, and so on. The program switches between these two modes at the press of a button.

Finally, *nlv* can serve as a simple front-end to the *Real-time Collector* agent. Through this interface, the user can actually request for the addition or subtraction of monitored events, such as CPU load or *ping*, on any host which is being managed by the JAMM system. Because the user may add their own monitoring tools to the JAMM system, this interactive visualization-driven monitoring may also include application-specific behaviors and tests, creating a very rich environment for exploring application performance characteristics.

3. JAMM Monitoring System

Historically, tools to monitor system characteristics such as CPU load and disk I/O statistics were included with NetLogger and were distributed as part of the NetLogger Toolkit. However, we have more recently come to the conclusion that the complex task of managing and automating the log data in a distributed system demands a complementary but distinct architecture. The architecture we have developed, called Java Agents for Monitoring and Management (JAMM) [ref], uses Java Remote Method Invocation (RMI) [ref] to launch a wide range of system and network monitoring tools and then extract, summarize, and publish the results. The JAMM components are illustrated in Figure 2.

Figure 2: JAMM Components

One of the primary goals of the JAMM system was to make the execution of sensors dependent on actual client usage of the distributed system, in other words to only perform monitoring "on-demand". On-demand monitoring reduces the total amount of data, thus reducing system perturbation and simplifying data management.

JAMM is based on a producer/consumer model, similar to the CORBA Event Service [ref]. Consumers contact an agent to subscribe to certain monitored events, which are then pushed back to the consumer from the event producer as a stream of data. A single query/response facility is also provided for consumers interested in only a single datum.

3.1. Sensors

The JAMM system is designed to control a collection of *sensors*. A sensor is any program which generates a time-stamped performance monitoring event. For example, there are sensors to monitor CPU usage, memory usage, network usage, and server status. Users can incorporate their own events into the system by wrapping their program with, or inheriting from, the provided Java class *JAMM.Sensor*.

3.2. Directory service

The directory service, currently implemented with the Lightweight Directory Access Protocol (LDAP)[ref], provides the location of all event supplier agents and sensors. This allows for look-up and discovery of all the monitoring available on the system.

3.3. Event supplier agent

At least one host in the distributed system must run the event supplier agent, which receives requests from consumers for data from particular sensor(s) on particlar host(s), communicates with the lower-level *sensor manager* to start and stop sensors when necessary, and updates entries in the directory service to reflect the current status of all sensors, along with some long-term summaries of their measurements.

3.4. SOAP

Consumers communicate with the event supplier agent using the Simple Object Access Protocol (SOAP). SOAP provides a generic XML-based form of remote procedure calls. The event supplier agent has a set of well-defined procedures (or "methods" in Java terminology) which provide the semantics of its interaction with the consumer.

4. Real-time Collector

The interface between the NetLogger application monitoring and visualization tools, and the system-level monitoring performed by systems such as JAMM, is called the *real-time collector*. The jobs of the real-time collector are to coordinate the aggregation of event trace data from applications with monitoring data, and also to provide a gateway to the monitoring management system for a real-time event consumer such as *nlv*. In order to do this, the real-time collector must perform three functions: receive application data, process SOAP requests, and translate between XML and ULM.

In this discussion, JAMM will be used as an example of a producer of system-level monitoring data. However, we anticipate that the adoption of SOAP and XML standards for monitoring data by several

members of the research community, particularly the Grid Forum Performance and Monitoring Working Group [ref], will spawn the development of compatible systems which perform a similar function.

When a consumer wishes to create a flat file containing both application event traces and JAMM monitoring data, they first start the real-time collector. The collector will automatically be able to receive application event logs, replicating the functionality of *netlogd* (described above). To add monitored system events, the consumer sends a SOAP request to the real-time collector indicating a monitored and event of interest. The real-time collector translates this request into a SOAP request for JAMM's *event supplier*. Data is then returned to the real-time collector directly from the JAMM sensors. This sequence is illustrated in Figure 3.

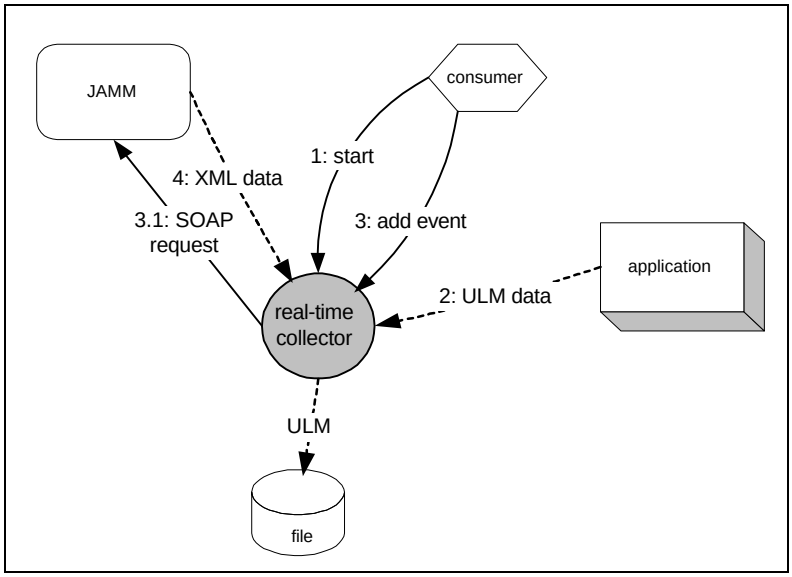

Figure 3: Real-time collector action sequence

Because the new standard for event logs is XML, and not ULM, the real-time collector may also need to translate the system-level monitoring input. However, the XML schema for these monitored events is well-known, making this task is straightforward in most cases.

5. Results

We will present experimental results of a NetLogger analysis of a wide-area distributed application, the Distributed Parallel Storage System (DPSS) [ref]. The analysis was motivated by unexpectedly low throughput on a testbed network between LBNL in Berkeley,CA and Ixx Sxx Ixx (ISI) in Washington, D.C. The visualization results provided a clearer picture of the bottlenecks in the system, and allowed for refinement of application and network parameters, resulting in higher application throughput. **I hope**

5.1. DPSS and NetLogger

The DPSS is essentially a "logical block" server whose functional components are distributed across a wide-area network. The DPSS uses parallel operation of distributed servers to supply high-speed data streams. Using off-the-shelf Unix workstations and disks, a typical four-server DPSS can deliver an aggregated data stream to an application of about 400 Mbits/s (50 MBytes/s). Other papers describing the DPSS in more detail [ref] are available from: http://www-didc.lbl.gov/DPSS/papers.html.

The DPSS and the client used in this discussion have been instrumented with the NetLogger API to produce time stamps for all important events. These time stamps follow the *data block* through the system, from client request to client receipt of the data. Figure 4 illustrates the instrumentation points in the DPSS architecture.

Figure 4: DPSS Performance Monitoring Points

5.2. Experiment

For a demonstration of the SuperMEMS project [ref], involving collaborators from ISI, MIT, Sarnoff Laboratories, UC Berkeley, and LBNL (**others**?), the DPSS was used to transfer data between a host in Berkeley,CA and Washington, D.C. Although the network connection between these hosts was a private (i.e. testbed) OC-12 link (622 Mbits/s), slightly above the theoretical maximum for the DPSS output, the observed throughput for the transfer was only about 30 Mbits/s. Fortunately for the project demonstration, tighter bottlenecks in other parts of the distributed application made this throughput sufficient. However, we thought that discovering the reason for this bottleneck would be an important and instructive task.

The DPSS was already instrumented with NetLogger calls, but for this experiment an extra instrumentation point was inserted into the client, *dpss_get* (which fetches one dataset specified from the command line). NetLogger timing information was output for every low-level *read()* from the incoming data stream(s).

All the DPSS servers and the receiving client were monitored for CPU load, available memory, TCP retransmits. In addition, disk I/O statistics (**which ones?**) were monitored on the DPSS servers.

5.3. Analysis

The *nlv* graph of a typical run of the experiment is shown in Figure 6. From this graph, we can infer several important pieces of information. Bla bla bla…

Figure 5: *nlv* Graph of Experimental Results

6. Related projects

7. Future work